Grid World

Markov Decision Processes & Reinforced Learning

Noah Prince, Rodney Shaghoulian, David Young

December 2015

Contents

1 Introduction

1.1 Submission Overview

This writeup summarizes the procedure and results of two methods for finding policies in MDPs as well as a free model reinforcement learning technique when the transition model and utilties are unknown. It also contains discussion of observed results and attempts to provide some insight and reflection on the behavior of implemented algorithms. This report was produced for course "CS-440: Artificial Intelligence" at University of Illinois Urbana Champaign.

1.2 Problem Definition.

Grid World MDP

Consider the following environment.

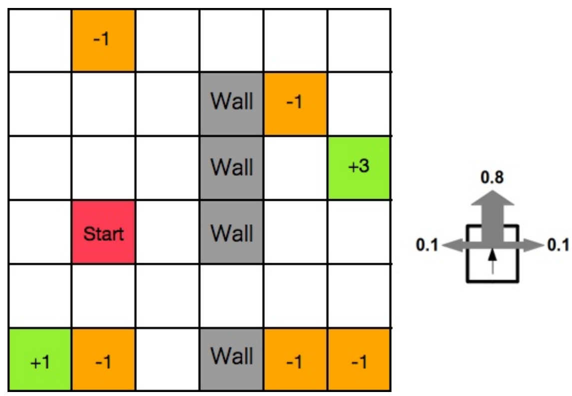


Figure 1: Environment

An agent is attempting to navigate the grid world beginning at the start tile. The transition model is as follows: the intended outcome (direction moved) occurs with probability 0.8, and with probability 0.1 the agent moves at either right angle to the intended direction (see the figure above). If the move would make the agent walk into a wall, the agent stays in the same place as before.

The rewards for the white squares are -0.04.

Assuming the known transition model and reward function listed above, find the optimal policy and the utilities of all the (non-wall) states using value iteration or policy iteration for two scenarios:

- Reward squares are treated as terminal states: once the agent reaches reward squares, either positive or negative, the agent stops moving.
- Reward squares are treated as non-terminal states: upon reaching reward squares, the agent continues to move. The agent's state sequence is infinite.

For each scenario, display the optimal policy and the utilities of all the states, and plot utility estimates as a function of the number of iterations as in Figure 17.5(a) (for value iteration, you should need no more than 50 iterations to get convergence). In this question and the next one, use a discount factor of 0.99.

Grid World Reinforcement Learning

Consider the reinforcement learning scenario in which the transition model and the reward function are unknown to the agent, but the agent can observe the outcome of its actions and the reward received in any state. (In the implementation, this means that the successor states and rewards will be given to the agent by some black-box functions whose parameters the agent doesn't have access to.)

Use Temporal Difference (TD) Q-Learning to learn an action-utility function only for the terminal scenario described above. Experiment with different parameters for the exploration function and report which choice works the best. For the learning rate function, start with $alpha(t) = \frac{60}{59+t}$, and play around with the numbers to get the best behavior.

Plot utility estimates and their RMS error (root mean squared error) as a function of the number of trials.

$$RMSE(U', U) = \sqrt{\frac{1}{N}\sum_{s}(U'(s) - U(s))^2} \tag{1}$$

where U'(s) is the estimated utility of state s, U(s) is the "true" utility as determined by value iteration, and N is the number of states.

2 Background

2.1 Markov Decision Processes

Hidden Markov Models involve a sequence of observations where one tries to reason about the underlying state sequence (ie: NO actions are involved). Markov decision processes however, involve actions that when taken (in turn) will affect the state of the world. For example, consider a game show where a contestant faces a series of questions of increasing difficulty and increasing payoff. After each question the contestant must decide whether to take the earnings and quit, or go for the next question with greater reward but higher chance of loosing everything.

Vocabulary

- **States:** s starting with s_0
- **Actions:** a each state s has actions $A(s)$ available to it.
- **Transition Models:** $P(s'|s,a)$ makes use of the Markov assumption: the probability of going to s' from s depends only on s and a and not on any other past actions or states.
- **Reward Functions:** $R(s)$
- **Policy:** π_s is the action that an agent takes in any given state. A policy is the "map" or "solution" to an MDP.
- **Markov Assumption:** the probability of going to s' from s depends only on s and a and not on any other past actions or states.

Solving MDPs
When presented with an MDP, the goal is often to find an optimal policy to navigate the MDP. An optimal policy should maximize the expected utility over all possible state sequences produced by following that policy. That is to say that $\sum P(sequence)U(sequence)$ over all state sequences starting from s_0 should be maximized. The utility of a state sequence $U(sequence)$ is the sum of rewards of individual states.

The actual utility of a state is hard to know without considering future actions. What is known is that the maximum utility is a result of taking optimal actions at every step. Optimal actions will yield a state s' with optimal utility, so select the action a that yields the maximum value for the expected utility of taking action a in state s. That is, the action a that maximizes the $\sum_{allpossibles'} P(s'|s,a) * U(s')$.

There are a lot more factors to consider. For example, to handle the possibility of infinite sequences, individual state rewards are discounted by an discount factor over time, such that sooner rewards count more than later rewards. Things tend to get recursive when dealing with future expectations, and there are many equations and algorithms to assist in developing optimal policies which map optimal actions to maximize utility. These equations and algorithms will be introduced in coming sections.

2.2 Bellman Equations

The bellman equation describes a recursive relationship between the utilities of successive states given a transition function. That is to say it is a recursive expression for $U(s)$ in terms of the utilities of its successor states. To describe the equation abstractly: The utility of state s is equal to sum of a reward function R(s) and the maximum expected utility of taking action a from state s considering all available actions. The expected utility of taking action a in state s is described by the summation in the formal equation below.

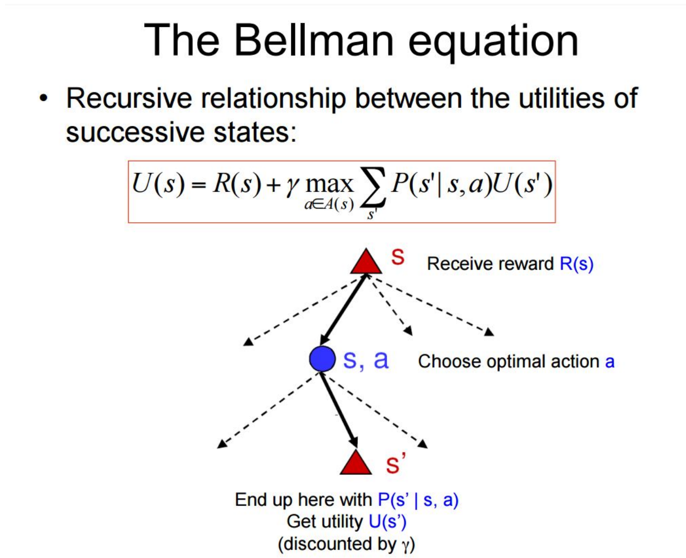


Figure 2: Bellman Equation.

For N states, there are N equations in N unknowns. Solving them will solve the MDP. Unfortunately, trying to solve them through expectimax search might run into trouble with infinite sequences. So instead, solve them algebraically using one of two methods: value iteration and policy iteration.

2.3 Value Iteration & Policy Iteration

Before it was mentioned that solving an MDP involved finding an optimal policy to navigate the MDP. This section provides background on two different techniques used to solve the simple MDP presented.

Value Iteration:
Value iteration is a tecnique for solving an MDP. This technique does not determine the policy directly, but instead determines the expected utility of each state, such that the movement policy can simply be read off the final grid of utility values.

- Start with every U(s) = 0
- Iterate until convergence
- During ith iteration, update utility of each state according to bellman equation. ie: set U_{i+1} using the bellman equation ontaining U_i

$$U_{i+1}(s) \Leftarrow R(s) + \gamma * max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s') \tag{2}$$

- In the limit of infinitely many iterations, the utility values are guaranteed to converge to the correct utility values. In practice, an infinite number of iterations is unnecessary.

Policy Iteration:
Policy iteration is a technique for solving an MDP. Policy iterations starts with some initial policy and then alternates between the following steps:

- **policy evaluation**: calculate the expected utility for every state s
- **policy improvement**: calculate new policy based on updated utilities. ie: populate the mapping from states to actions that defines the policy using the formula

$$Policy^{i+1}(s) = argmax_{a \in A(s)} \sum_{s'} P(s'|s,a) * updatedUtilityForSJustCalculated \tag{3}$$

2.4 Reinforcement Learning

With a regular MDP, the transition model and reward functions are known. Solving such an MDP involves finding the optimal policy to navigate the space. In some cases however, the transition model and reward function are initially unknown. In such cases, reinforcement learning can "learn" the right policy by "doing". However, this necessitates taking actions as the agent learns this policy. The basic Reinforcement Learning scheme can be outlined as follows. At each time step:

- Take an action
- Observe outcome (successor state & reward)
- Update internal representation of environment and policy
- If you reach a terminal state, just start over (new trial)

Model-Based vs. Model-Free

There are two types of reinforcement learning. The first is called Model-Based. In Model-Based reinforcement learning, the agent learns the model of the MDP and tries to solve the MDP concurrently. The model here refers to the transition probabilities and rewards. With Model-Based reinforcement learning, the actual "learning" involves:

- keeping track of how many times state s' follows state s when action a is taken, and update the transition probability $P(s'|s, a)$ according to the relative frequencies (observed)
- keeping track of the rewards $R(s)$

Once the model is learned, using it is similar to the regular MDP problems previously described. Ie: Estimate the utilities $U(s)$ using Bellman's equations.

Alternatively, Model-Free reinforcement learning involves learning how to act without explicitly learning the transition probabilities $P(s'|s, a)$. Q-Learning is an example of Model-Free reinforcement learning. In Q-Learning, the agent learns an action-utility function $Q(s, a)$ that tells the value of doing action a in state s.

Convergence:

Theoretically, the utility estimates and policy will converge to the same results that value and policy iteration provides. However, since the agent is operating in an environment where the transition model and rewards are unknown, it will need more iterations and trials through the maze before results will converge.

Exploration vs. Exploitation:
Exploration refers to taking a new action with unknown consequences. Exploration can discover higher reward states that what has previously been found, yielding a more accurate model of the environment. However exploration can also lead to negative encounters and utility is not being maximized while exploring. Exploitation on the other hand refers to selecting the best action found so far. Exploitation can maximize rewards as reflected in the current utility estimates and will avoid negative encounters. Unfortunately exploitation will often fail to discover truly optimal strategies.

To achieve convergence, the agent needs to balance exploration vs. exploitation. In the initial trials through the maze, the agent should try to explore to find high-reward states. The agent tries each direction from each state a number of times equal to some threshold value. As the estimates of Q-values get more accurate, the agent should exploit more, meaning it's actions are selected based off which direction has the highest Q value. To summarize, the agent should explore more in the beginning and become more greedy over time.

Temporal Difference Q-Learning:

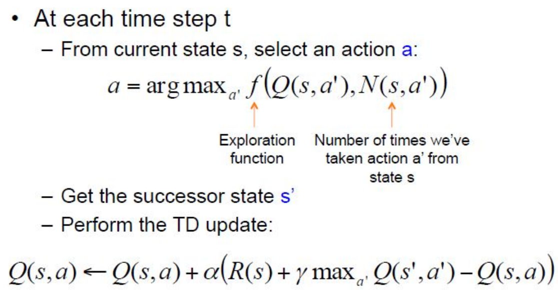


Figure 3: TD Q-Learning Overview.

As previously mentioned, Q-Learning is a type of Model-Free reinforcement learning. The main difference in Reinforcement Learning, as compared to the methods used for Value/Policy iteration, is that 1) The transition model, and 2) the rewards associated with each state, are not known to us. With a Model-Free approach, instead of taking actions to learn the transition model (as in Model-Based), the agent learns an action-utility function $Q(s, a)$ that tells the value of doing action a in state s. In the end the agent won't need to know the transition model. In a normal MDP, the policy would be defined as the action that gives the max value of the sum of $P(s'|s, a)U(s')$ for all possible s'. But now with Q-Learning the policy is just defined as the action that yields the max $Q(s, a)$.

2.5 Grid World

The following diagrams help to visualize the simple grid world that is the subject of this paper's implementation.

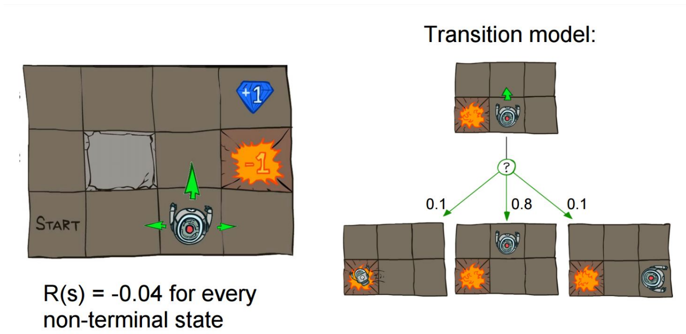


Figure 4: GridWorld Definition.

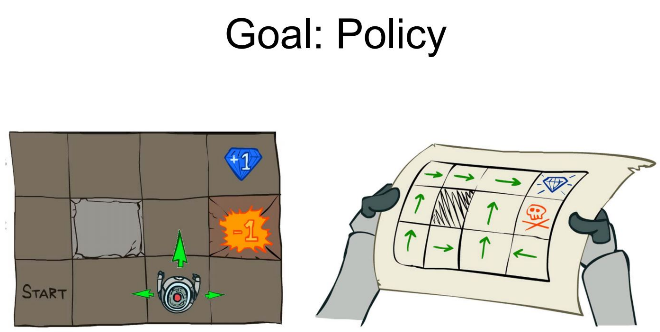


Figure 5: GridWorld Goal: Policy.

3 Overview of Source

Obtaining the source code
The entirety of the code written for this project can be found at the following repository:

https://github.com/noahprince22/GridWorldMDP/

Summary of source code
The following source files were written from scratch. All code is well commented with Javadocs; it should be no burden to browse for specific details.

Filename	**Description**
Direction.java	Enum for movement directions.
DrawingBoard.java	Draws the a simple graphical representation of the gridworld.
GridSquare.java	Holds information for a grid space in the gridworld.
GridWorld.java	Holds the state of the grid world and performs value and policy iteration.
GridWorld_Q.java	Extends GridWorld, and performs Q learning.
Main.java	Entry point to run 1.1.
Main2.java	Entry point to run 1.2.

Additionally, a small third party 2D graphics library was included (StdDraw.java) to provide graphics used in DrawingBoard.java.

4 Implementation: Solving MDPs

4.1 Representing States and Actions

A gridSquare class was created to handle the representation of both states and actions. In the grid world, a state is merely a space on the grid and its associated values. Since any action results in the transition to a state, it is possible to represent an action with the state that it leads to. Therefore, both states and actions are represented with the gridSquare class.

4.2 Representing the GridWorld

The grid world is represented as an array of gridSpaces. The GridWorld constructor replicates the grid world environment as outlined in the problem definition, by specifying the wall and reward attributes of appropriate gridSpaces.

4.3 Getting the Max Utility Action for a Square

Throughout the policy and value iterations, it is often necessary to get the maximum possibility utility action for a square. This follows the formula given in lecture:

$$\underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s'|s,a)U(s') \tag{4}$$

Key to this implementation is the ability to get the utility from a given action (current square to another gridsqaure). This code accomplishes that:

```
// Action here is defined as one gridspace to another
public double getUtilityForAction(GridSquare currentState, GridSquare successorState) {
    if (currentState == null || successorState == null)
        return 0;

    double utility;

    int x = currentState.getxPos();
    int y = currentState.getyPos();

    GridSquare leftSquare  = getGridSquare(x - 1, y);
    GridSquare rightSquare = getGridSquare(x + 1, y);
    GridSquare downSquare  = getGridSquare(x, y + 1);
    GridSquare upSquare    = getGridSquare(x, y - 1);

    if (isValidLocation(successorState) && ! successorState.isWall())
        utility = 0.8 * successorState.utility;
    else
        utility = 0.8 * currentState.utility; // agent stays in same place as before.

    if (successorState.getxPos() == currentState.getxPos()) { // intended movement is
        vertical
        if (isValidLocation(leftSquare) && ! leftSquare.isWall())
            utility += 0.1 * leftSquare.utility;
        else
            utility += 0.1 * currentState.utility; // agent stays in same place as before.

        if (isValidLocation(rightSquare) && ! rightSquare.isWall())
            utility += 0.1 * rightSquare.utility;
        else
            utility += 0.1 * currentState.utility; // agent stays in same place as before.
    }
    else { // intended movement is horizontal
        if (isValidLocation(upSquare) && ! upSquare.isWall())
            utility += 0.1 * upSquare.utility;
        else
            utility += 0.1 * currentState.utility; // agent stays in same place as before.

        if (isValidLocation(downSquare) && ! downSquare.isWall())
            utility += 0.1 * downSquare.utility;
        else
            utility += 0.1 * currentState.utility; // agent stays in same place as before.
    }

    return utility;
}
```

The following java code calculates the formula using the previous function and returns the GridSquare which is the state the max action would lead to.

```java
public GridSquare maxUtilityAction(GridSquare s) {
    List<GridSquare> validNextSquares = getValidAdjacentSquares(s);
    if (validNextSquares.size() != 0) {
        GridSquare maxNode = validNextSquares.get(0);
        double maxUtility = Double.NEGATIVE_INFINITY;

        // For all valid directions to move, get the node with the expected max utility
        for (GridSquare s_prime : validNextSquares) {
            double utility = getUtilityForAction(s, s_prime);

            if (utility > maxUtility) {
                maxNode = s_prime;
                maxUtility = utility;
            }
        }

        return maxNode;
    }
    else
        return null;
}
```

Along with this function is the helper function maxUtility, which gets the utility from the maxUtilityAction:

```java

public double maxUtility(GridSquare s) {
    GridSquare maxAction = maxUtilityAction(s);

    if (maxAction != null)
        return getUtilityForAction(s, maxAction);
    else
        return 0;
}
```

4.4 Value Iteration

Value iteration occurs numIterations times. For each value iteration, when the square isn't a wall, we calculate it's utility as:

$$\text{Reward of This Square} + \text{discountFactor} * \text{maxUtility}(square) \tag{5}$$

After a GridWorld is initialized, the following code from GridWorld.java performs value iteration to establish utilties for all the grid spaces such that a policy can be determined.

```java
/* We must not write the utilities to cells until all 36 are calculated */
public void establishValueIterationUtilities() {
    for (int i = 1; i <= numIterations; i++) {
        double utilities[][] = new double[rows][columns];
        /* 1st calculate all the utilities before writing to the cells */
        for (int y = 0; y < rows; y++) {
            for (int x = 0; x < columns; x++) {
                GridSquare currentSquare = getGridSquare(x, y);
                if ( ! currentSquare.isWall())
                    utilities[y][x] = currentSquare.getReward() + discountFactor *
                        maxUtility(currentSquare);
            }
        }
        /* Now we can copy the utilities */
        for (int y = 0; y < rows; y++) {
            for (int x = 0; x < columns; x++) {
                grid[y][x].utility = utilities[y][x];
            }
        }
    }
}
```

4.5 Policy Iteration

A policy iteration consists of two parts: Policy Evaluation and Policy Improvement. In policy evaluation, we update the utilities for all squares according to the following equation. With this equation, we are updating the utilities according to the utility gained by our current policy

Utility(s) = reward(s) + discountFactor * Utility for the Action Defined by policy(s)

In policy improvement, we change our policy so that all actions are the actions that produce the most utility under the curent model.

After a GridWorld is initialized, the following code from GridWorld.java performs the policy iteration numIteration times.

```java
public void policyEvaluation() {
    double utilities[][] = new double[rows][columns];
    /* 1st calculate all the utilities before writing to the cells */
    for (int y = 0; y < rows; y++) {
        for (int x = 0; x < columns; x++) {
            GridSquare currentSquare = getGridSquare(x, y);
            if ( ! currentSquare.isWall())
                utilities[y][x] = currentSquare.getReward() + discountFactor *
                    getUtilityForAction(currentSquare, policy[y][x]);
        }
    }
    /* Now we can copy the utilities */
    for (int y = 0; y < rows; y++) {
        for (int x = 0; x < columns; x++) {
            grid[y][x].utility = utilities[y][x];
        }
    }
}

public void policyImprovement() {
    for (int y = 0; y < rows; y++) {
        for (int x = 0; x < columns; x++) {
            policy[y][x] = maxUtilityAction(getGridSquare(x, y));
        }
    }
}

public void establishPolicyIterationUtilites() {
    // Setup initial policy (always go to first in adjacent list)
    for (int y = 0; y < rows; y++) {
        for (int x = 0; x < columns; x++) {
            GridSquare currentSquare = getGridSquare(x, y);
            List<GridSquare> validAdjacentSquares = getValidAdjacentSquares(currentSquare)
                ;

            if (validAdjacentSquares.size() == 0 )
                policy[y][x] = null;
            else
                policy[y][x] = validAdjacentSquares.get(0); // policy starts off with all
                    left-pointing arrows
        }
    }

    for (int i = 0; i < numIterations; i++) {
        policyEvaluation();
        policyImprovement();
    }
}
```

4.6 Retrieving the Policy from the calculated Utilities

The policy is continuously updated throughout the iterations. After the iterations have completed, the instance policy variable contains the optimal policy. This policy, in terms of the optimal grid square to move to from the current grid square, can then be analysed.

Analyzing this policy means translating the policy into directions to move at each grid. This is done via the generateDirectionPolicy function which uses the getDirection helper function for every node in the policy. Getting the direction from our definition of actions just means figuring out which direction the action square is relative to the current square. This is done as follows:

```java
public Direction getDirection(int row, int column, GridSquare destination){
    if (destination == null)
        return null;
    if (column - 1 == destination.getxPos())
        return Direction.LEFT;
    else if (column + 1 == destination.getxPos())
        return Direction.RIGHT;
    else if (row + 1 == destination.getyPos())
        return Direction.DOWN;
    else if (row - 1 == destination.getyPos())
        return Direction.UP;
    return null; // should never execute
}
```

5 Implementation: Reinforcement Learning ("Q-Learning")

5.1 Modifications to GridSquare

For the reinforcement Q-Learning, the following fields were added to GridSquare.

```
//q values for the 4 possible actions
public double qValueLeft;
public double qValueRight;
public double qValueUp;
public double qValueDown;

//counters for the number of times a given action has been taken
public int actionCounterLeft;
public int actionCounterRight;
public int actionCounterUp;
public int actionCounterDown;
```

The following methods were added to GridSquare as well.

```
public void updateUtility(){
    utility = Math.max(qValueLeft, Math.max(qValueRight, Math.max(qValueUp, qValueDown)));
}

public Direction highestUtilityDirection(){
    if (qValueLeft > Math.max(qValueRight, Math.max(qValueUp, qValueDown)))
        return Direction.LEFT;
    else if (qValueRight > Math.max(qValueUp, qValueDown))
        return Direction.RIGHT;
    else if (qValueUp > qValueDown)
        return Direction.UP;
    else
        return Direction.DOWN;
}

public Direction leastTriedDirection(){
    if (actionCounterLeft < Math.min(actionCounterRight, Math.min(actionCounterUp,
        actionCounterDown)))
        return Direction.LEFT;
    else if (actionCounterRight < Math.min(actionCounterUp, actionCounterDown))
        return Direction.RIGHT;
    else if (actionCounterUp < actionCounterDown)
        return Direction.UP;
    else
        return Direction.DOWN;
}
```

5.2 Initializing the GridWorld

The new fields and constructor for GridWorld_Q are shown below. The new GridWorld_Q class extends the previous GridWorld.

```java
int threshold;
GridSquare[][] solutionGrid;

public GridWorld_Q(boolean rewardsTerminal, int iterations, double discountFactor, int threshold, GridSquare[][] solutionGrid) {
    super(rewardsTerminal, iterations, discountFactor);
    this.threshold = threshold;
    this.solutionGrid = solutionGrid;
}
```

5.3 Establishing Q Values

```java
public void establish_Q_Utilities() {
    //while(notConverged()){ // can try this instead of the for loop line below
    for (int i = 1; i <= numIterations; i++){
        GridSquare currentState = start;
        GridSquare intendedSuccessorState = null;
        GridSquare actualSuccessorState = null;
        while (true){
            if(currentState.isTerminal()){
                currentState.utility = currentState.getReward();
                break;
            }
            intendedSuccessorState = selectAction(currentState);//this is an action
            Direction intendedDirection = getDirection(currentState, intendedSuccessorState);
            actualSuccessorState = getSuccessorState(currentState, intendedSuccessorState, intendedDirection);
            TD_Update(currentState, actualSuccessorState, intendedDirection);
            updateOtherVariables(currentState, intendedDirection);
            currentState = actualSuccessorState;
        }
    }
    policyImprovement();
    generateDirectionPolicy();
}
```

5.4 Selecting an Action

When selecting an action, we explore before exploiting. For each state, we first try to explore. If any of the actions from the specific state have been tried less than threshold=500 number of times, we try that action. If multiple actions have been tried less than 500 times, we pick the action with the fewest previous attempts. If the previous trial count of all actions exceeds threshold (500), we select the action that yields the highest Q(s,a), which corresponds to ”exploiting”.

5.5 Getting the Actual Successor State

Because the agent does not know the transition function it must take actions based off the intended outcome. In reality, an action results in one of three possible movements according to a probability function. To simulate this real probability function, each time the agent makes a move the code uses a virtual dice roll to select the actual state resulting from an action. A random number generator is used such that the intended successor state is chosen with probability 0.8, and the left and right states are chosen with probabilities 0.1 each.

5.6 TD-Update

After doing the action, we need to update Q(s,a). We use the following formula to update Q(s,a), while taking into account the learning rate. Nothing tricky here.

$$Q_{new}(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma max_{a'} Q(s',a') - Q(s,a)) \tag{6}$$

6 Results: Terminal Reward States

For the Value Iteration and Policy Iteration with Terminal Reward States, 50 iterations were performed. Looking at the utility estimate plots as a function of the number of iterations, one can see this converges.

6.1 Value Iteration

Optimal Policy

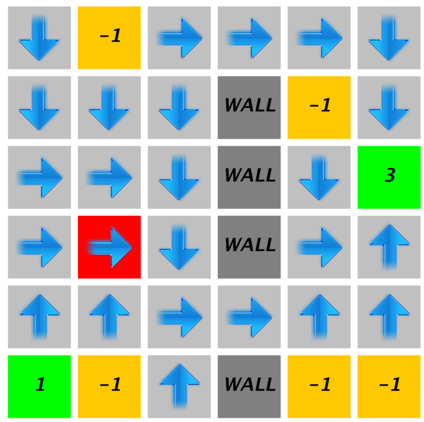


Figure 6: Optimal Policy for terminal case, post value iteration.

Utilities of All States

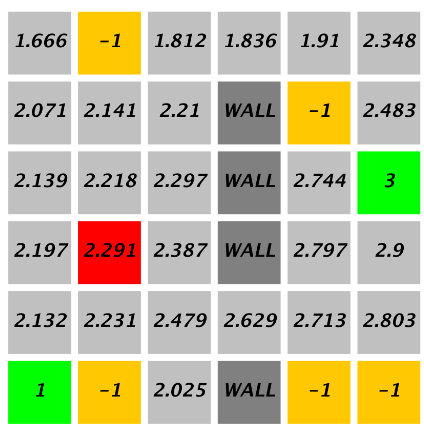


Figure 7: Utilities for terminal case, post value iteration.

Plot of utility estimates as a function of the number of iterations

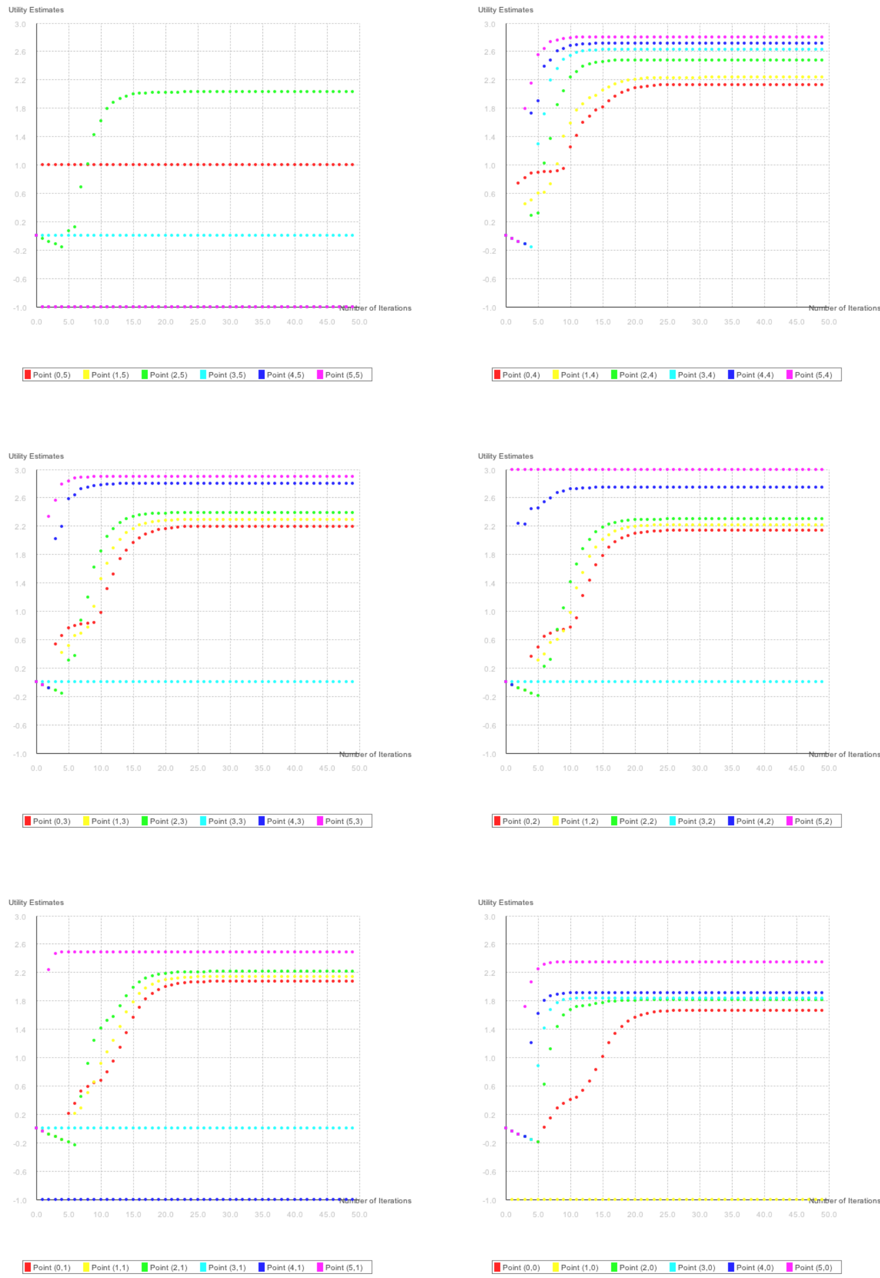


Figure 8: Utility Estimates vs Number of Iterations (Terminal Case)

6.2 Policy Iteration

Optimal Policy

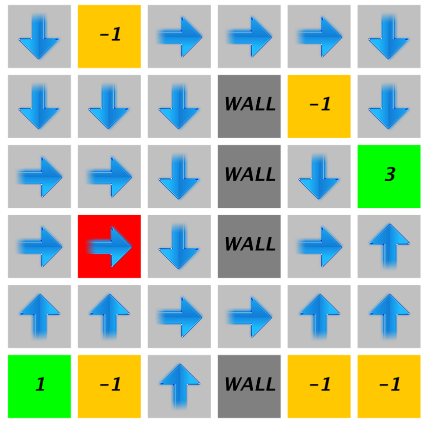


Figure 9: Optimal Policy in the terminal Case, post policy iteration.

Utilities of All States

1.666	-1	1.812	1.836	1.91	2.348
2.071	2.141	2.21	WALL	-1	2.483
2.139	2.218	2.297	WALL	2.744	3
2.197	2.291	2.387	WALL	2.797	2.9
2.132	2.231	2.479	2.629	2.713	2.803
1	-1	2.025	WALL	-1	-1

Figure 10: Utilities in terminal case, post policy iteration.

Plot of utility estimates as a function of the number of iterations

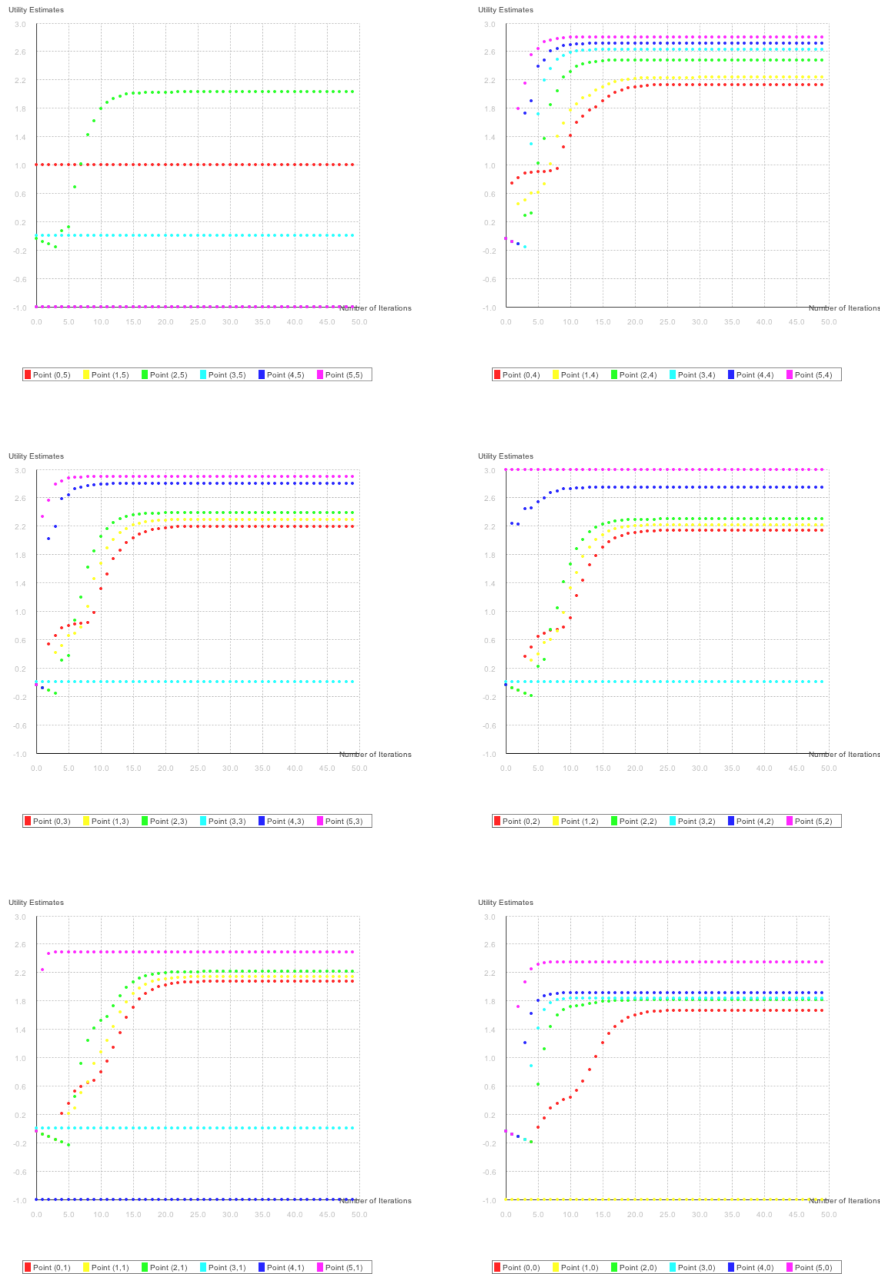


Figure 11: Utility Estimates vs Number of Iterations (Terminal Case)

7 Results: Non-Terminal Reward States

For the Value Iteration and Policy Iteration with Non-Terminal Reward States, 1000 iterations were performed. Looking at the utility estimate plots as a function of the number of iterations, one can see this converges.

7.1 Value Iteration

Optimal Policy

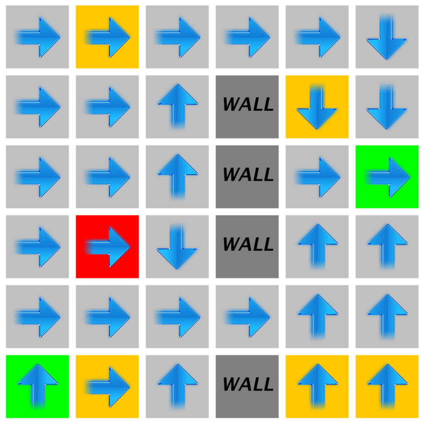


Figure 12: Optimal Policy for non-terminal case, post value iteration.

Utilities of All States

210.7	213.6	217.7	220.9	223.8	226.4
209.1	211.8	214.6	WALL	225.7	229.6
207.3	209.5	211.6	WALL	229.4	233.1
209	211.7	214.4	WALL	226.8	229.8
210.9	214	217.5	221.1	223.9	226.5
209.7	210.8	214.3	WALL	220.1	222.2

Figure 13: Utilities for non-terminal case, post value iteration.

Plot of utility estimates as a function of the number of iterations

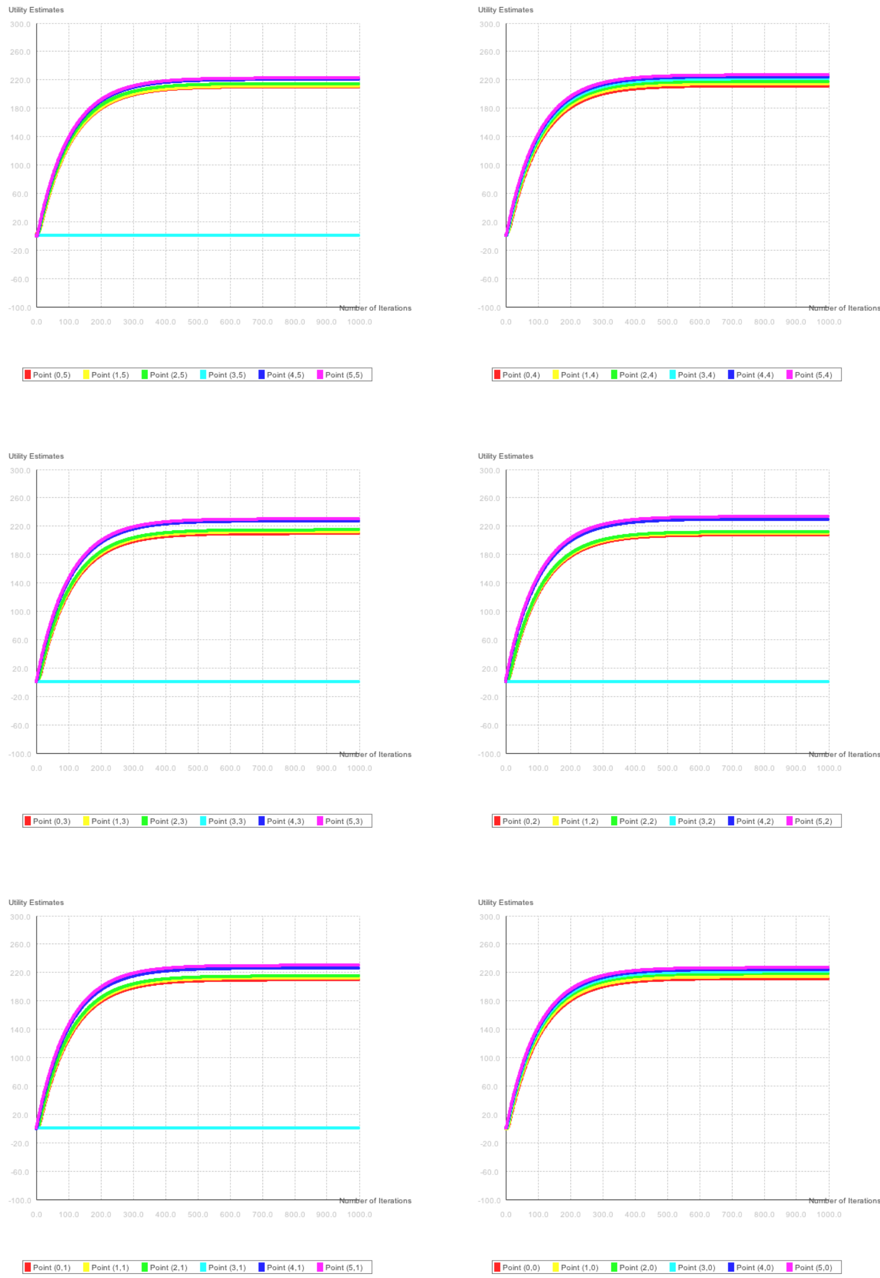


Figure 14: Utility Estimates vs Number of Iterations (Non-Terminal Case)

7.2 Policy Iteration

Optimal Policy

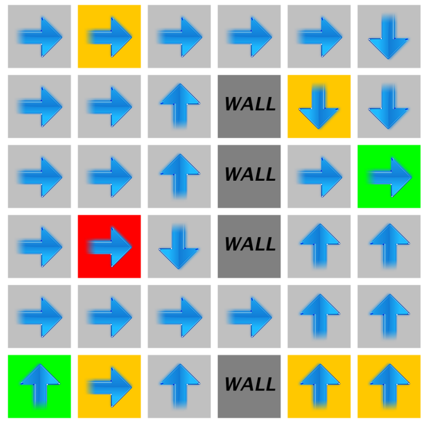


Figure 15: Optimal Policy in the non-terminal Case, post policy iteration.

Utilities of All States

210.7	213.6	217.7	220.9	223.8	226.4
209.1	211.8	214.6	WALL	225.7	229.6
207.3	209.5	211.6	WALL	229.4	233.1
209	211.7	214.4	WALL	226.8	229.8
210.9	214	217.5	221.1	223.9	226.5
209.7	210.8	214.3	WALL	220.1	222.2

Figure 16: Utilities in the non-terminal case, post policy iteration.

Plot of utility estimates as a function of the number of iterations

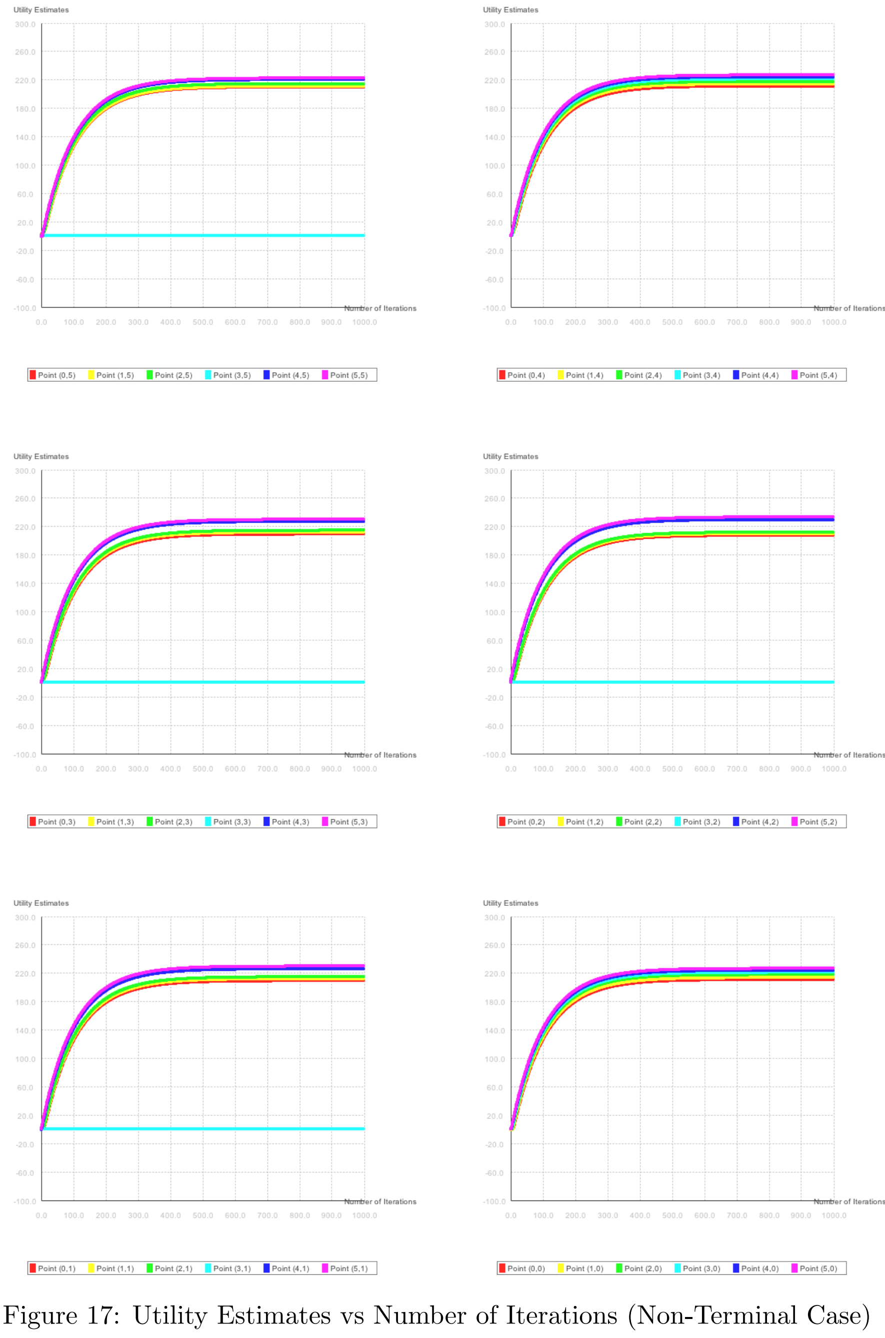
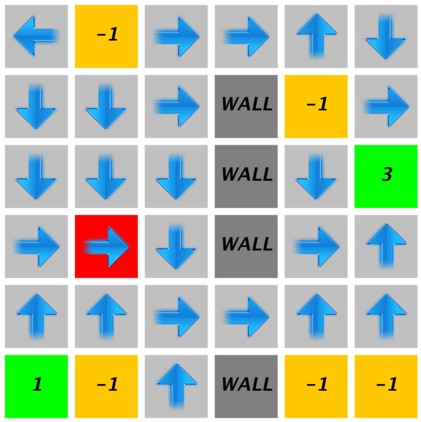


Figure 17: Utility Estimates vs Number of Iterations (Non-Terminal Case)

8 Results: Q-Learning

Utility Estimates



1.471	-1	1.594	2.118	2.668	2.86
1.535	1.193	2.253	WALL	-1	2.93
2.118	2.193	2.27	WALL	2.741	3
2.185	2.272	2.37	WALL	2.796	2.902
2.119	2.213	2.464	2.629	2.714	2.808
1	-1	1.873	WALL	-1	-1

Figure 18: Final Policy (left) and Utility Estimates (right) after Q-Learning

RMS error as a function of the number of trials
Looking at the RMSE plot, it is clear that the error decreases extremely quickly as the number of trials increases. This was expected as major fluctuations occur early in the process.

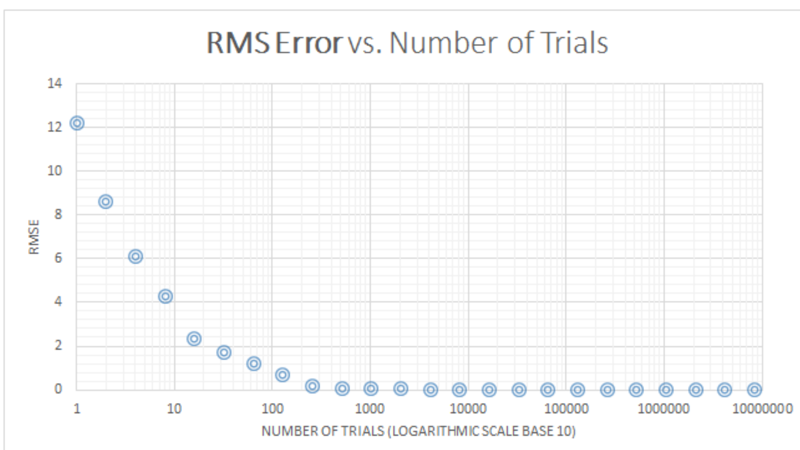


Figure 19: RMS error as a function of the number of trials

Utility plots

Each plot corresponds to the states in a specific row. The general trend is that the utilities converge. For row 0, it is more difficult to see the convergence since we don't try actions from that row as much as from the middle rows. The policy gives us the path to the highest reward state has more accurate utilities (which converge faster) since we try those actions a large number of times. We don't try row 0 actions quite as many times so our plot for row 0 has a slower convergence than the other rows.

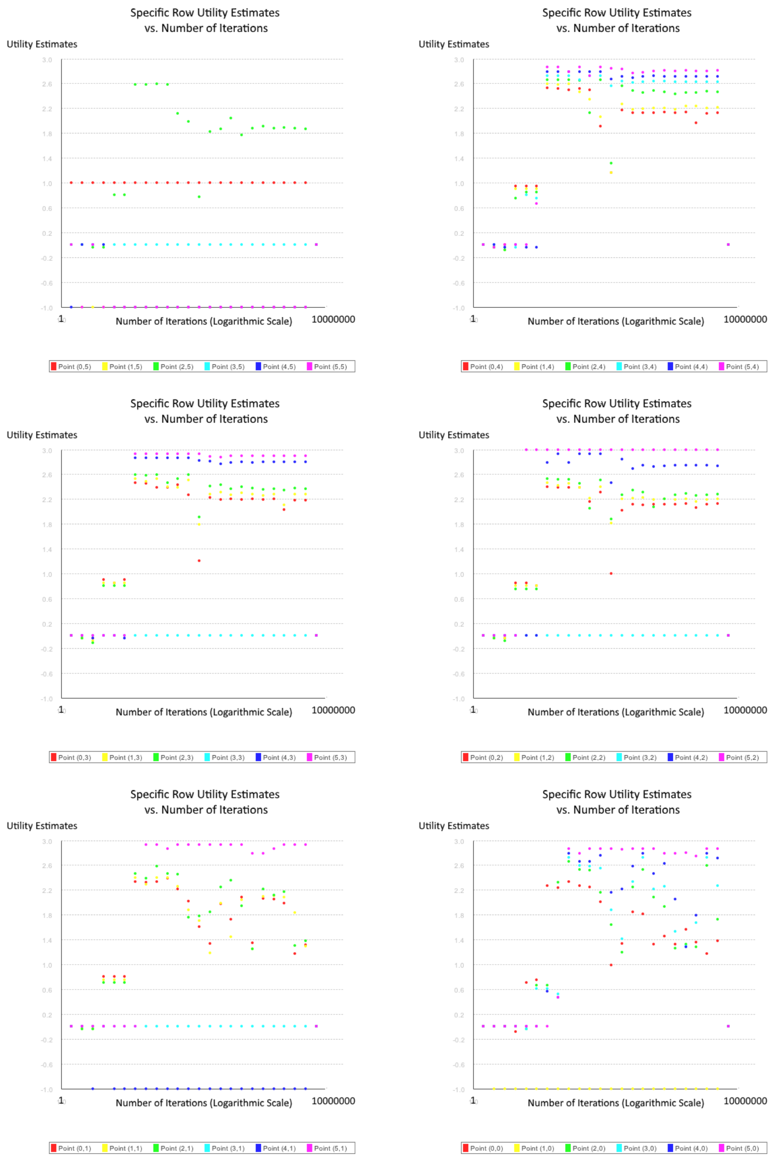


Figure 20: Utility Estimates vs Number of Iterations

Experimenting with Different Parameters
There are two main parameters that we tune to get different results:

1. Exploration Function - Specifically, by changing the threshold value of the number of times we try each direction from a state, we can specify how much exploration we want our agent to do. We currently use threshold = 500, which is a fair amount of exploration before we try exploiting.

2. Learning Rate - Our "t" value for alpha is done slightly differently than the method presented in lecture. We tie our "t" value to each action from a state (i.e. there are 4 "t" values for each state). "t" will range from 1 to 500, so for learning rate, we used a modified formula of $\frac{600}{599+t}$. This learning rate value decays more steadily since t can be as large as 500. Other learning rates such as $\frac{60}{59+t}$ and $\frac{6000}{5999+t}$ were attempted but resulted in slower convergence.

9 Analysis & Discussion

Both the policy and value iteration worked as expected. Looking at the optimal policy, one can see that the tendency is to head for green goal states as quickly as possible.

An interesting result from the non terminal reward state game is that the policy is to run into a wall/edge of the map while on positive reward nodes. This is taking advantage of the fact that moving into a wall counts as a move. It is to the agent's benefit to stay in a reward state as long as possible.

Note that the game actively fights this strategy in that there is only an 80% chance the agent will remain in that reward state.

With the non-terminal reward states utility plot, one can see that the utility for each node is ever-rising, only halted by the discount factor. This is because the agent can continue to reap rewards for every iteration. One can see this process in the plots with the increasing of the expected utility. Another consequence of the non-terminal utilities was that many more iterations were needed before convergence. The number of iterations needed for convergence would be stunted with a higher discount factor; the discount factor is what causes the non-terminal reward state games to converge.

With the terminal reward state utilities, one can see that the plots level off much sooner. This is because the agent achieves a higher utility by stopping in a reward state, not by moving around; this is because moving around is penalized by -.04.

The policy iteration and value iteration results ended up being the same. This was an expected consequence of calculating the optimal utilities.

9.1 Potential Improvements

An improvement on our search algorithm for 1.2 would be to do Policy Search instead of Q Learning. Instead of getting the exact Q-values right, we can simply get our ordering right. To do this, we would write down the policy as a function of some paramters and adjust the parameters to improve the expected reward.

References